Overview

• Pointers
  • Variables that hold memory addresses
  • Using pointers to do “call-by-reference” in C

• Arrays
  • List of elements of the same type
  • Array name is alias for the address of the first array element
  • Array name can be used as a constant pointer

• Strings (next lecture)
  • Array of characters ending in ‘\0’
  • Special functions for manipulating strings
Exercise: Write a swap function

```c
void swap(int x, int y) {
    ______________
    ______________
    ______________
    ______________
}

int main(void) {
    int a = 11, b = 22;
    swap(a,b);
    printf("a=%d, b=%d\n");
}
```

Output should be a=22, b=11

Memory Allocation

- Each time a variable is defined, memory space is set aside for it. The amount of memory depends on the type of the variable. Examples:
  
<table>
<thead>
<tr>
<th>Type</th>
<th>Memory Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1 byte</td>
</tr>
<tr>
<td>int</td>
<td>2 or 4 or 8 bytes (machine dependent)</td>
</tr>
</tbody>
</table>

- The memory space is accessed using the variable name
Computer Memory

Function Parameters

- C always passes arguments BY-VALUE
  - Memory allocated for parameters when function is called
  - Argument VALUES copied into newly allocated memory
  - Memory automatically de-allocated when function ends

```c
void swap(int x, int y)
{
    __________
    __________
    __________
    __________
}

int main(void) {
    int a = 11, b = 22;
    swap(a, b);
    printf("a=%d, b=%d\n");
}
```
Visualizing Memory Space

```c
void swap(int x, int y) {
    ______________
    ______________
    ______________
    ______________
}

int main(void) {
    int a = 11, b = 22;
    swap(a, b);
    printf("a=%d, b=%d\n");
}
```

Main Memory

What are Pointers?

- A pointer is a variable that holds the ADDRESS of another variable
- Suppose that we have an integer variable
  ```c
  int i;
  ```
  and wish to have a pointer point to this variable.
  ```c
  int *p = &i;
  ```
- How do we know where `i` is located?

  `&i` is the address of `i`. The operator `&` is called the ADDRESS-OF operator.
Pointers and Addresses

• We can declare that a pointer iPtr points to an int by saying

\[
\text{int } * \text{iPtr;}
\]

• Suppose that we have:
  \[
  \text{int } i = 5;
  \]
  \[
  \text{int } j = 7;
  \]

• We can make iPtr point to \(i\) by assigning to iPtr the memory location where \(i\) is stored. Thus

\[
i\text{Ptr} = &i;
\]

sets iPtr to point to \(i\).

Declaring Pointers

• When declaring several pointer variables in one statement - the asterisk does not distribute throughout the statement:

\[
\text{int } * p, q;
\]

equivalent to

\[
\text{int } * p;
\text{int } q;
\]

\[
\text{int } * p, * q;
\]

equivalent to

\[
\text{int } * p;
\text{int } * q;
\]
Initializing Pointers

- We can also initialize `iPtr` at the point of declaration:

  ```c
  int i;
  int * iPtr = &i;
  ```

- Here is a common error:

  ```c
  int i;
  int * iPtr = i;  // ERROR: i is not an address
  ```

Dereference *

- The value of the data being pointed at is obtained by using the operator

- If `p` is a pointer value, then

  ```c
  *p
  ```

  refers to the variable pointed to by `p`. Since reference is another name for address, the operator `*` is called dereference operator.
Dereference Example

A dereferenced pointer behaves exactly like the variable it points to.

Example Program

```c
#include <stdio.h>
int main(void) {
    char x = 'M';
    char* p = &x;
    printf("Value of x is %c\n", x);
    printf("Address of x is %x\n", p);
    printf("Address of p is %x\n", &p);
    return 0;
}
```

What is the output?

What is this?

*p
swap Function Revisited

```c
void swap(int *x, int *y)
{
    __________
    __________
    __________
    __________
}

int main(void) {
    int a = 11, b = 22;
    swap(____, ____);
    printf("a=%d, b=%d\n");
}
```

Output should be:  \texttt{a=22, b=11}

Note the Difference ...

Assume:

\begin{itemize}
\item \hspace{1cm} ptr1 \quad i \quad 5
\item \hspace{1cm} ptr2 \quad j \quad 7
\end{itemize}

After

\begin{itemize}
\item \hspace{1cm} ptr1 = ptr2;
\end{itemize}

\begin{itemize}
\item \hspace{1cm} ptr1 \quad i \quad 7
\item \hspace{1cm} ptr2 \quad j \quad 7
\end{itemize}

After

\begin{itemize}
\item \hspace{1cm} *ptr1 = *ptr2;
\end{itemize}
Uninitialized Pointers

- Suppose that we have the following declarations:

```
int i;
int * iPtr;
*iPtr = 100;
```

- What is the value of `iPtr`? Undefined. What could happen?
  - `iPtr` could hold an address that does not make sense at all, causing your program to crash if dereferenced.
  - `iPtr` could point to an address which is accessible. Then the assignment `*iPtr = 100;` would accidentally change some other data, which could result in a crash at a later point. This is a tough error to detect since the cause and symptom may be widely separated in time.

Putting it all Together...

```
int i, value;
int * iPtr; // declares iPtr to be a pointer to an integer
i = 510;    /* Step 1 */
iPtr = &i;  /* Step 2 */
value = *iPtr; /* Step 3 */
```

<table>
<thead>
<tr>
<th>iPtr</th>
<th>i</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>510</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>510</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>100</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
The null Pointer

• The value of a pointer can be:
  • some garbage (pointer unassigned)
  • the address of some variable (eg., int * p = &i; )
  • the constant 0 (the null pointer, points to absolutely nothing)

  ```c
  somePointer = 0;
  ```

  *This statement does not cause somePointer to point to memory location zero; it guarantees that somePointer does not point to anything.*

• The null pointer is a special pointer value that a program can test for:

  ```c
  if (somePointer == 0) ... 
  ```

Arrays vs. Pointers
Arrays and Pointers

• An array name is basically a constant pointer

\[ a, \&a[0] \rightarrow 1000 \]
\[ \&a[1] \rightarrow 1004 \]
\[ \&a[2] \rightarrow 1008 \]

• Consider the declaration:

\[ \text{int } a[3]; \]

• The compiler allocates three integers for the array object. These are referenced as \( a[0], a[1], a[2] \) and occupy a contiguous block of memory.

• The value of \( a \) is exactly \( \&a[0] \), the address of the first integer in the array.

Array Indices

• Logically, valid indices for an array range from 0 to \( \text{MAX}-1 \), where \( \text{MAX} \) is the dimension of the array

\[
\begin{align*}
\text{int } a[6]; \\
\text{stands for} \\
a[0], a[1], a[2], a[3], a[4] \text{ and } a[5] \\
\text{Logically, there is no } a[6]!
\end{align*}
\]

• Memory allocation
Arrays: C vs. Java

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Arrays</strong></td>
<td>int [] a = new int [10];</td>
<td>int a[10];</td>
</tr>
<tr>
<td></td>
<td>float [][] b = new float [5][20];</td>
<td>float b[5][20];</td>
</tr>
<tr>
<td><strong>Array bound checking</strong></td>
<td>// run-time check</td>
<td>/* no run-time check */</td>
</tr>
</tbody>
</table>

C Does Not Do Bounds Checking!

```plaintext
int a[5];
int a[10];
int *a = (int*)malloc(10*sizeof(int));
float b[5][20];
float *b = (float*)malloc(5*20*sizeof(float));
```

Unpleasant if you happened to have another variable before the array variable `a`, or after it!
Hands-On: Active Learning

- Write a small program to test what happens when you use invalid array indices.

Clobbering Example

```c
/* This program accesses an invalid array cell. ** Why does it work? Draw the memory map. */
int a[2];    /* 2 cells, each cell 4 bytes (32 bits) */
int b[4];    /* 4 cells, each cell 4 bytes (32 bits) */
int c[4];    /* 4 cells, each cell 4 bytes (32 bits) */
char d[5];   /* 5 cells, each cell 1 byte (8 bits) */

main()
{
    a[0]=5;
    b[1]=4;
    c[0]=9;
    d[4] = 'a';

    b[4]=10;
    printf("%d\n",b[4]);
    printf("%d\n",c[0]);   /* Why did c[0] change? */
}
```
Memory Space

<table>
<thead>
<tr>
<th>Address</th>
<th>Label</th>
</tr>
</thead>
<tbody>
<tr>
<td>400</td>
<td>a[0]</td>
</tr>
<tr>
<td>404</td>
<td>a[1]</td>
</tr>
<tr>
<td>408</td>
<td>b[0]</td>
</tr>
<tr>
<td>412</td>
<td>b[1]</td>
</tr>
<tr>
<td>416</td>
<td>b[2]</td>
</tr>
<tr>
<td>420</td>
<td>b[3]</td>
</tr>
<tr>
<td>424</td>
<td>c[0]</td>
</tr>
<tr>
<td>428</td>
<td>c[1]</td>
</tr>
</tbody>
</table>

NO Aggregate Array Operations

• Aggregate operations refer to operations on an array as a whole, as opposed to operations on individual array elements.

```c
#define MAX 100
int x[MAX];
int y[MAX];
```

• There are no aggregate operations on arrays:

  - Assignment: `x = y;` Error!
  - Comparison: `if (x == y) ...` Error!
  - I/O: `printf("%d", x);` Error!
  - Arithmetic: `x = x + y;` Error!
Hands-On

• Write a small program that uses aggregate array operations. What error messages do you get?

Arrays vs. Pointers
Arrays and Pointers - Examples

• Consider the following declarations:

```c
int a[5] = {1, 2, 3};
int * p;
```

```c
p = &a[2];
```

Indexing can be used with any pointer, but it only makes sense when the pointer points to an array.

Arrays are NOT Pointers

• Declaring an array sets aside space for its elements

```c
char a[5];
```

• Declaring a pointer variable sets aside only space to hold the variable

```c
char * p;
```

• You can change a pointer variable, but not the address of an array

```c
char b[6];
p = b;   // OK
b = p;   // ERROR !
```
Indexing Pointers

```c
int a[5];
int *p, *q;
p = a;
p[1] = 44;
q = p + 2;
q[-1] = 43;
q[2] = 46;
```

Pointer Arithmetic

```c
int a[5];
p = a;
p = a + 2;
```

Subscript: $a[i]$ “means” $(a+i)$

`int *p;`  `p = a + 2;`

Note: arithmetic scales by data size (e.g., int of 4 bytes)
Quaint usage of pointer arithmetic

Add up the elements of an array:

```c
int a[100];
int sum, *p;
...
for (p=a; p<a+100; p++)
    sum += *p;
```

More straightforwardly:

```c
int a[100];
int sum, i;
...
for (i=0; i<100; i++)
    sum += a[i];
```

Array Parameters to Functions

```c
void printArray(int *p, int n) {
    int i;
    for (i=0; i<n; i++)
        printf("%d\n",p[i]);
}

int fib[5] = {1, 1, 2, 3, 5};

int main(void) {
    printArray(fib, 5);
}
```
Array Params \equiv Pointer Params

```c
void printArray(int *p, int n) { ... }
void printArray(int p[5], int n) { ... }
void printArray(int p[], int n) { ... }
void printArray(int p[1000], int n) { ... }
```

All these declarations are equivalent!
Try them out.

```c
int main(void) {
    printArray(fib, 5);
}
```

Exercise: Reverse Array

- Reverse the values in an array
  - Inputs: integer array \( a \), and number of elements \( n \)
  - Output: values of \( a \) stored in reverse order
- Algorithm
  - Swap the first and last elements in the array
  - Swap the second and second-to-last elements
  - ...

```
77 31 94 5 186
```
Summary

• C variables
  • Pointers
  • Arrays

• Required Readings
  • Chapter 4.7 of the textbook